# Theoretical Computational Linguistics: Finite-state Automata

Jeffrey Heinz

March 31, 2022

# Contents

# Chapter 1

# Introduction: Strings and Trees

## 1.1 Computational Linguistics: Course Overview

In this class, we will study:

1. Automata Theory
2. Haskell
3. . . . as they pertain to problems in linguistics:
   (a) *Well-formedness* of linguistic representations
   (b) *Transformations* from one representation to another

### 1.1.1 Linguistic Theory

Linguistic theory often distinguishes between *well- and ill-formed representations.*

**Strings.** In English, we can coin new words like *bling*. What about the following?

1. *gding*
2. *θwɪk*
3. *spɪf*

**Trees.** In English, we interpret the compound *deer-resistant* as an adjective, not a noun. What about the following?

1. *green-house*
2. *dry-clean*
3. *over-throw*

Linguistic theory is often also concerned with *transformations*.

**Strings.** In generative phonology, underlying representations of words are *transformed* to surface representations of words.

1. /kæt-z/ → [kæts]
2. /wɪʃ-z/ → [wɪʃɪz]

**Trees.** In derivational theories of generative syntax, the deep sentence structure is *transformed* into a surface structure.

1. Mary won the competition.
   (a) The competition was won by Mary.
   (b) What did Mary win?

### 1.1.2 Automata Theory

Automata are *abstract* machines that answer questions like these.

**The Membership Problem**

**Given:** A possibly infinite set of strings (or trees) $X$.

**Input:** A input string (or tree) $x$.

**Problem:** Does $x$ belong to $X$?

**The Transformation Problem**

**Given:** A possible infinite function of strings to strings (or trees to trees) $f : X \to Y$.

**Input:** A input string (or tree) $x$.

**Problem:** What is $f(x)$?

There are many kinds of automata. Two common types of automata address these specific problems.

**Recognizers** Recognizers solve the membership problem.
**Transducers** Transducers solve the transformation problem.

Different kinds of automata instantiate different kinds of memory.

**Finite-state Automata** An automata is finite-state whenever the amount of memory necessary to solve a problem for input $x$ is fixed and **independent** of the size of $x$.

**Linear-bounded Automata** An automata is linear-bounded whenever the amount of memory necessary to solve a problem for input $x$ is **bounded by a linear function** of the size of $x$.

In this class we will study finite-state recognizers and transducers. There are many types of these as well, some are shown below.

- deterministic vs. non-deterministic

- 1way vs. 2way (for strings)
- bottom-up vs. top-down vs. walking (for trees)

The simplest type is the deterministic, 1way recognizer for strings. We will start there and then complicate them bit by bit:

1. add non-determinism
2. add output (transducers)
3. add 2way-ness
4. generalize strings to trees and repeat

What do automata mean for linguistic theory?

**Fact 1:** Finite-state automata over strings are sufficient for phonology (Johnson, 1972; Kaplan and Kay, 1994).

**Fact 2:** Finite-state automata over strings are *NOT* sufficient for syntax, but linear-bounded automata are (Chomsky, 1956; Shieber, 1985, among others).

**Fact 3:** Finite-state automata over trees *ARE* sufficient for syntax (Rogers, 1998; Kobele, 2011; Graf, 2011).

**Hypothesis:** Linguistic phenomena can be modeled with special kinds of finite-state automata with even stricter memory requirements over the right representations (Graf, 2015; Heinz, 2018).

Strings and Trees
In this section, we define strings and trees of finite size inductively.

### 1.1.3 Strings

Informally, strings are sequences of symbols.

What are symbols? It is standard to assume a set of symbols called the *alphabet*. The Greek symbol $\Sigma$ is often used to represent the alphabet but people also use $S$, $A$, or anything else. The symbols can be anything: IPA letters, morphemes, words, part-of-speech categories. $\Sigma$ can be infinite in size, but we will usually consider it to be finite.

Formally, strings are defined inductively with an operation called *concatenation*. Concatenation is an operation, like addition, which build new strings from existing ones. For now, we will write it as "$\cdot$". *Inductive definitions* are defined in two parts: the *base case* and an *inductive case*.

**Definition 1** (Strings, version 1)**.**

**Base Case:** *If $a \in \Sigma$ then $a$ is a string.*
**Inductive Case:** *If $a \in \Sigma$ and $w$ is a string then $a \cdot (w)$ is a string.*

**Example 1.** Let $\Sigma = \{a, b, c\}$. Then the following are strings.

1. $a \cdot (b \cdot (c))$
2. $a \cdot (a \cdot (a))$
3. $a \cdot (b \cdot (c \cdot (c)))$

Frankly, writing all the parentheses and "$\cdot$" is cumbersome. So the above examples are much more readable if written as follows.

1. *abc*
2. *aaa*
3. *abcc*

Writing all the parentheses and "$\cdot$" is also unnecessary because the above definition provides a unique "derivation" for each string.

**Example 2.** Let $\Sigma = \{a, b, c\}$. We claim $w = a \cdot (b \cdot (c \cdot (c)))$ is a string. There is basically one way to show this. First we observe that $a \in \Sigma$ so whether $w$ is a string depends on whether $x = b \cdot (c \cdot (c))$ is a string by the inductive case. Next we observe that since $b \in \Sigma$ whether $x$ is a string depends on whether $y = c \cdot (c)$ is a string, again by the inductive case. Once more, since $c \in \Sigma$ whether $y$ is a string depends on whether $c$ is a string. Finally, by the base case $c$ is a string and so the dominoes fall: $y$ is a string so $x$ is a string and so $w$ is a string.

This unique derivabality is useful in many ways. For instance, suppose we want to determine the length of a string. Here is how we can do it.

**Definition 2** (string length)**.** *The* length *of a string $w$, written $|w|$, is defined as follows. If there is $a \in \Sigma$ such that $w = a$ then $|w| = 1$. If not, then $w = a \cdot (x)$ where $x$ is some string and $a \in \Sigma$. In this case, $|w| = |x| + 1$.*

Note length is an inductive definition!

**Example 3.** What is the length of string $w = abcc$? Well, as before we see that $w = a \cdot x$ where $x = bcc$. Thus, $|w| = 1 + |bcc|$. What is the length of *bcc*? Well, $x = b \cdot y$ where $y = cc$. So now we have $|w| = 1 + (1 + |cc|)$. Since $y = c \cdot z$ where $z = c$ we have $|w| = 1 + (1 + (1 + |c|))$. Finally, by the *base case* we have $|w| = 1 + (1 + (1 + (1))) = 4$.

There is another way to define strings, which makes use of the so-called empty string. The empty string is usually written with one of the Greek letters $\epsilon$ or $\lambda$. It's just a matter of personal preference. The empty string is useful from a mathematical perspective in the same way the number zero is useful. Zero is a special number because for all numbers $x$ it is the case that $0 + x = x + 0 = x$. The empty string serves the same special purpose. It is the unique string with the following special property with respect to concatenation.[1]

$$\text{For all strings } w,\ \lambda \cdot w = w \cdot \lambda = w \tag{1.1}$$

With this concept, we can redefine strings as the following.

[1] Technically, concatenation is not yet defined between strings since it is only defined for a string and a symbol. Our goal would be to ensure the property mentioned holds once concatenation is defined between strings.

**Definition 3** (Strings, version 2)**.**

**Base Case:** *$\lambda$ is a string.*
**Inductive Case:** *If $a \in \Sigma$ and $w$ is a string then $a \cdot (w)$ is a string.*

As a technical matter, when we write the string *abc*, we literally mean the following structure: $a \cdot (b \cdot (c \cdot (\lambda)))$.

The definition for length can be redefined similarly.

We can now define concatenation between strings. First we define ReverseAppend, which takes two strings as arguments and returns a third string.

**Definition 4** (reverse append)**.** Reverse append *is a binary operation over strings, which we denote $\otimes_{\texttt{revapp}}$. You can also think of it as a function which takes two strings $w_1$ and $w_2$ as arguments and returns another string. Here is the base case. If $w_1 = \lambda$ then it returns $w_2$. So we can write $\lambda \otimes_{\texttt{revapp}} w = w$. Otherwise, there is $a \in \Sigma$ such that $w_1 = a \cdot (x)$ for some string $x$. In this case,* reverse append *returns $x \otimes_{\texttt{revapp}} a \cdot (w_2)$.*

**Exercise 1.** Work out what $abc \otimes_{\texttt{revapp}} def$ equals.

**Exercise 2.** What is $abc \otimes_{\texttt{revapp}} \lambda$? Write a definition for string reversal.

**Exercise 3.** Define the concatenation of two strings $w_1$ and $w_2$ using reverse append and string reversal. Prove this definition satisfies Equation 1.1.

The set of all strings of finite length from some alphabet $\Sigma$, including the empty string, is written $\Sigma^*$. A *stringset* is a subset of $\Sigma^*$.

Stringsets are often called *formal languages.* From a linguistic perspective, is is the study of string well-formedness.

### 1.1.4 Trees

Trees are like strings in that they are recursive structures. Informally, trees are structures with a single 'root' node which dominates a sequence of trees.

Formally, trees extend the dimensionality of string structures from 1 to 2. In addition to linear order, the new dimension is dominance.

Unlike strings, we will not posit "empty" trees because every tree has a root.

Like strings, we assume a set of symbols $\Sigma$. This is sometimes partitioned into symbols of different types depending on whether the symbols can only occur at the leaves of the trees or whether they can dominate other trees. We don't make such a distinction here.

**Definition 5** (Trees)**.** *If $a \in \Sigma$ and $w$ is a string of trees then $a[w]$ is a tree.*

A tree $a[\lambda]$ is called a *leaf.* Note if $w = \lambda$ we typically write $a[\,]$ instead of $a[\lambda]$. Similarly, if $w = t_1 \cdot (t_2 \cdot (\ldots \cdot (t_n \cdot (\lambda)) \ldots))$, we write $a[t_1 t_2 \ldots t_n]$ for readability.

The definition of trees above may appear circular. It appears circular since it defines trees in terms of strings of trees. However, this circularity is an illusion. The definition has a solid recursive base case, as I will now explain. The key to resolving this illusion is to construct the full set of trees in steps. For example, $\lambda$ is a string (of trees) by the definition of string. With the empty string $\lambda$ and the finite alphabet $\Sigma$ we can define a set of trees $\mathrm{T}_0 = \{a[\lambda] \mid a \in \Sigma\}$. $\mathrm{T}_0$ is the set of all logically possible leaves (trees of depth 0). $\mathrm{T}_0$ is a finite alphabet and so $\mathrm{T}_0^*$ is a well defined set of strings over this alphabet. For example $w = a[\ ] \cdot (b[\ ] \cdot (c[\ ] \cdot (b[\ ] \cdot (\lambda))))$ is a string of trees.

So far, with $\Sigma$ and $\lambda$ we built $\mathrm{T}_0$. The definition of strings gives us $\mathrm{T}_0^*$. Now with $\Sigma$ and $\mathrm{T}_0^*$ we can build $\mathrm{T}_1 = \{a[w] \mid a \in \Sigma, w \in \mathrm{T}_0^*\} \cup \mathrm{T}_0$. $\mathrm{T}_1$ includes $\mathrm{T}_0$ in addition to all trees of depth 1. With $\mathrm{T}_1$, and the definition of strings we have $\mathrm{T}_1^*$. Now with $\Sigma$ and $\mathrm{T}_1^*$ we can build $\mathrm{T}_2 = \{a[w] \mid a \in \Sigma, w \in \mathrm{T}_1^*\} \cup \mathrm{T}_1$.

More generally, we define $\mathrm{T}_{n+1} = \{a[w] \mid a \in \Sigma, w \in \mathrm{T}_n^*\} \cup \mathrm{T}_n$. Finally, let the set of all logically possible trees be denoted with $\Sigma^{\mathrm{T}} = \bigcup_{i \in \mathbb{N}} \mathrm{T}_i$. Figure 1.1 illustrates this construction.

Figure 1.1: The inductive definition of trees.

$$\begin{tikzcd}
 & \Sigma \arrow[d] & \text{str.def.} \arrow[d] & \Sigma \arrow[d] & \text{str.def.} \arrow[d] & \Sigma \arrow[d] & & \Sigma \arrow[d] & \text{str.def.} \arrow[d] & \\
\lambda \arrow[r] & \mathrm{T}_0 \arrow[r] & \mathrm{T}_0^* \arrow[r] & \mathrm{T}_1 \arrow[r] & \mathrm{T}_1^* \arrow[r] & \mathrm{T}_2 \arrow[r] & \cdots \arrow[r] & \mathrm{T}_n \arrow[r] & \mathrm{T}_n^* \arrow[r] & \mathrm{T}_{n+1}
\end{tikzcd}$$

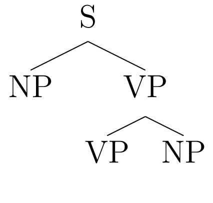
Here are some examples of trees.

**Example 4.** Let $\Sigma = \{\mathrm{NP}, \mathrm{VP}, \mathrm{S}\}$. Then the following are trees.

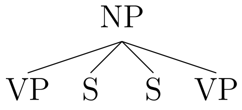
1. S[ NP[ ] VP[ VP[ ] NP[ ] ] ]
2. NP[ VP[ ] S[ ] S[ ] VP[ ] ]
3. NP[ NP[ NP[ ] VP[ ] S[ ] ] ]

We might draw these structures as follows.

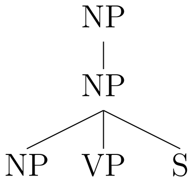
(1)



(2)



(3)



Regarding the tree in (1), its leaves are NP, VP, and NP.

As before, we can now write definitions to get information about trees. For instance here is a definition which gives us the number of nodes in the tree.

**Definition 6.** *The* size *of a tree $t$, written $|t|$, is defined as follows. If there is some $a \in \Sigma$ such that $t = a[\ ]$ then its size is* 1. *If not, then $t = a[t_1 t_2 \ldots t_n]$ where $a \in \Sigma$ and each $t_i$ is a tree. Then $|t| = 1 + |t_1| + |t_2| + \ldots + |t_n|$.*

**Exercise 4.** Using the above definition, calculate the size of the trees (1)-(3) above. Write out the calculation explicitly.

Here is a definition for the *width* of a tree.

**Definition 7.** *The* depth *of a tree t, written* $\textbf{\texttt{depth}}(t)$*, is defined as follows. If there is some* $a \in \Sigma$ *such that* $t = a[\ ]$ *then its depth is* 0*. If not, then* $t = a[t_1 t_2 \ldots t_n]$ *where* $a \in \Sigma$ *and each* $t_i$ *is a tree. Then* $\textbf{\texttt{depth}}(t) = 1 + \texttt{max}\big\{\texttt{depth}(t_1), \texttt{depth}(t_2), \ldots, \texttt{depth}(t_n)\big\}$ *where* `max` *takes the largest number in the set.*

**Definition 8.** *The* width *of a tree t, written* $\textbf{\texttt{width}}(t)$*, is defined as follows. If there is some* $a \in \Sigma$ *such that* $t = a[\ ]$ *then its width is* 0*. If not, then* $t = a[t_1 t_2 \ldots t_n]$ *where* $a \in \Sigma$ *and each* $t_i$ *is a tree. Then* $\textbf{\texttt{width}}(t) = \texttt{max}\big\{n, \texttt{width}(t_1), \texttt{width}(t_2), \ldots, \texttt{width}(t_n)\big\}$ *where* `max` *takes the largest number in the set.*

The set of trees $\Sigma^{\mathrm{T}}$ contains all trees of arbitrary width. Much research also effectively concerns the set of all and only those trees whose width is bounded by some number $n$. Let $\Sigma^{\mathrm{T}(n)} = \{t \in \Sigma^{\mathrm{T}} \mid \texttt{width}(t) \leq n\}$.

**Exercise 5.** The *yield* of a tree $t$, written $\texttt{yield}(t)$, maps a tree to a string of its leaves. For example let $t$ be the tree in (1) in Example 4 above. Then its yield is the string "NP VP NP".

### 1.1.5 String Exercises

**Exercise 6.** Let $\Sigma$ be the set of natural numbers. So we are considering strings of numbers.

1. Write the definition of the function *addOne* which adds one to each number in the in the string. So *addOne* would change the string $5 \cdot (11 \cdot (4 \cdot (\lambda)))$ to $6 \cdot (12 \cdot (5 \cdot (\lambda)))$. Using this definition, show *addOne* of the following number strings is calculated.

   (a) $11 \cdot (4 \cdot (\lambda))$

   (b) $3 \cdot (2 \cdot (\lambda))$

   (c) $\lambda$

2. Write the definition of the *timesTwo* of the numbers in the string. So *timesTwo* would change the string $5 \cdot (11 \cdot (4 \cdot (\lambda)))$ to $10 \cdot (22 \cdot (8 \cdot (\lambda)))$. Using this definition, show *timesTwo* of the following number strings is calculated.

   (a) $11 \cdot (4 \cdot (\lambda))$

   (b) $3 \cdot (2 \cdot (\lambda))$

   (c) $\lambda$

**Exercise 7.** Let $\Sigma$ be the set of natural numbers. So we are considering strings of numbers.

1. Write the definition of the *sum* of the numbers in the string. Using this definition, show how the sum of the following number strings is calculated.

   (a) $11 \cdot (4 \cdot (\lambda))$

   (b) $3 \cdot (2 \cdot (\lambda))$

   (c) $\lambda$

2. Write the definition of the *product* of the numbers in the string. Using this definition, show how the sum of the following number strings is calculated.

   (a) $11 \cdot (4 \cdot (\lambda))$

   (b) $3 \cdot (2 \cdot (\lambda))$

   (c) $\lambda$

### 1.1.6 Tree Exercises

**Exercise 8.** Let $\Sigma$ be the set of natural numbers. Now let's consider trees of numbers.

1. Write the definition of the function *addOne* which adds one to each number in the tree. Using this definition, calculate *addOne* as applied to the trees below.

   (a) 4[ 12[ ] 3[ ] ]

   (b) 4[ 12[ ] 3[ 1[ ] 2[ ] ] ]

   (c) 4[ 12[ 7[ ] 7[ 6[ ] ] ] 3[ 1[ ] 2[ ] ] ]

2. Write the definition of the function *isLeaf* which changes the nodes of a tree to `True` if it is a leaf node or to `False` if it is not. Using this definition, calculate *isLeaf* as applied to the trees below.

   (a) 4[ 12[ ] 3[ ] ]

   (b) 4[ 12[ ] 3[ 1[ ] 2[ ] ] ]

   (c) 4[ 12[ 7[ ] 7[ 6[ ] ] ] 3[ 1[ ] 2[ ] ] ]

**Exercise 9.** Let $\Sigma$ be the set of natural numbers. Now let's consider trees of numbers.

1. Write the definition of the *sum* of the numbers in the tree. Using this definition, show how the sum of the following number trees is calculated.

   (a) 4[ 12[ ] 3[ ] ]

   (b) 4[ 12[ ] 3[ 1[ ] 2[ ] ] ]

   (c) 4[ 12[ 7[ ] 7[ 6[ ] ] ] 3[ 1[ ] 2[ ] ] ]

2. Write the definition of the *yield* of the numbers in the string. The yield is a string with only the leaves of the tree in it. Using this definition, calculate the yields of the trees below.

    (a) 4[ 12[ ] 3[ ] ]

    (b) 4[ 12[ ] 3[ 1[ ] 2[ ] ] ]

    (c) 4[ 12[ 7[ ] 7[ 6[ ] ] ] 3[ 1[ ] 2[ ] ] ]

# Chapter 2

# String Acceptors

## 2.1 Deterministic Finite-state String Acceptors

### 2.1.1 Orientation

This section is about deterministic finite-state acceptors for strings. The term *finite-state* means that the memory is bounded by a constant, no matter the size of the input to the machine. The term *deterministic* means there is single course of action the machine follows to compute the output from some input. As we will see later, *non-deterministic machines* can be thought of as pursuing multiple computations simultaneously. The term *acceptor* is synonymous with *recognizer*. It means that this machine solves *membership problems*: given a set of objects $X$ and input object $x$, does $x$ belong to $X$? The term *string* means we are considering the membership problem over stringsets. So $X$ is a set of strings (so $X \subseteq \Sigma^*$) and the input $x$ is a string.

### 2.1.2 Definitions

**Definition 9.** *A* deterministic finite-state acceptor (DFA) *is a tuple* $(Q, \Sigma, q_0, F, \delta)$ *where*

- $Q$ *is a finite set of states;*
- $\Sigma$ *is a finite set of symbols (*the alphabet*);*
- $q_0 \in Q$ *is the* initial *state;*
- $F \subseteq Q$ *is a set of* accepting *(*final*) states; and*
- $\delta$ *is a function with domain* $Q \times \Sigma$ *and co-domain* $Q$*. It is called the* transition function.

We extend the domain of the transition function to $Q \times \Sigma^*$ as follows. In these notes, the empty string is denoted with $\lambda$.

$$\begin{aligned} \delta^*(q, \lambda) &= q \\ \delta^*(q, aw) &= \delta^*((\delta(q, a), w) \end{aligned} \tag{2.1}$$

Consider some DFA $A = (Q, \Sigma, q_0, F, \delta)$ and string $w \in \Sigma^*$. If $\delta^*(q_0, w) \in F$ then we say $A$ *accepts/recognizes/generates* $w$. Otherwise $A$ *rejects* $w$.

**Definition 10.** *The stringset recognized by* $A$ *is* $L(A) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}$.

The use of the 'L' denotes "Language" as stringsets are traditionally referred to as *formal languages*.

**Definition 11.** *A stringset is* regular *if there is a DFA that recognizes it.*

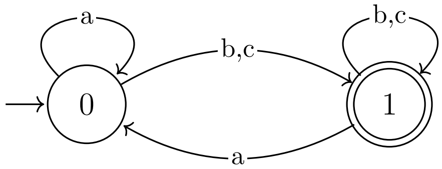
### 2.1.3 Exercises

**Exercise 10.** This exercise is about designing DFA. Let $\Sigma = \{a, b, c\}$. Write DFA which express the following generalizations on word well-formedness.

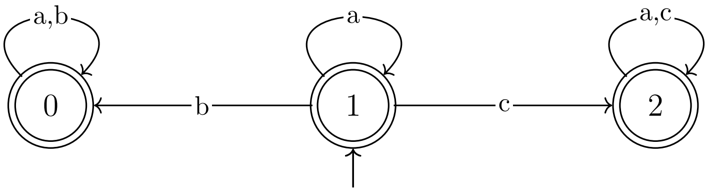
1. All words begin with a consonant, end with a vowel, and alternate consonants and vowels.
2. Words do not contain *aaa* as a substring.
3. If a word begins with *a*, it must end with *c*.
4. Words must contain two *b*s.
5. All words have an even number of vowels.

**Exercise 11.** This exercise is about reading and interpreting DFA. Provide generalizations in English prose which accurately describe the stringset these DFA describe.

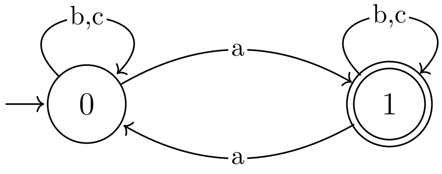
1.



2.



3.



4. Write the DFA in #1-3 in mathematical notation. So what is $Q, \Sigma, q_0, F$, and $\delta$?

## 2.2 Properties of DFA

Note that for a DFA $A$, its transition function $\delta$ may be partial. That is, there may be some $q \in Q, a \in \Sigma$ such that $\delta(q, a)$ is undefined. If $\delta$ is a partial function, $\delta^*$ will be also. It is assumed that if $\delta^*(q_0, w)$ is undefined, then $A$ rejects $w$.

We can always make $\delta$ total by adding one more state to $Q$. To see how, call this new state $\diamondsuit$. Then for each $(q, a) \in Q \times \Sigma$ such that $\delta(q, a)$ is undefined, define $\delta(q, a)$ to equal $\diamondsuit$. Every string which was formerly undefined w.r.t. to $\delta^*$ is now mapped to $\diamondsuit$, a non-accepting state. This state is sometimes called the *sink* state or the *dead* state.

**Definition 12.** *A DFA is* complete *if $\delta$ is a total function. Otherwise it is* incomplete*.*

It is possible to write DFA which have many useless states. A state can be useless in two ways. First, there may be no string which forces the machine to transition into the state. Second, there may be a state from which no string

**Definition 13.** *A state $q$ in a DFA $A$ is* useful *if there is a string $w$ such that $\delta^*(q_0, w) = q$ and a string $v$ such that $\delta^*(q, v) \in F$. Otherwise $q$ is* useless*. If every state in $A$ is useful, then $A$ is called* trim*.*

Not all complete DFAs are trim. If there is a sink state, it is useless in the above sense of the word.

**Definition 14.** *A DFA $A$ is* minimal *if there is no other DFA $A'$ such that $L(A) = L(A')$ and $A'$ has fewer states than $A$.*

Technically, not all complete DFAs are minimal. If there is a sink state, it is not minimal.

**Exercise 12.** Consider the DFAs in the exercise 11. Are they complete? Trim? Minimal?

## 2.3 Some Closure Properties of Regular Languages

A set of objects $X$ is *closed* under an operation $\circ$ if for all objects $x, y \in X$ it is the case that $x \circ y \in X$ too.

We can easily show that the union of any two regular stringsets $R$ and $S$ is also regular. Let $A_R = (Q_R, \Sigma, q_{0R}, F_R, \delta_R)$ be the DFA recognizing R and let $A_S = (Q_S, \Sigma, q_{0S}, F_S, \delta_S)$ be the DFA recognizing S. We can assume $A_R$ and $A_S$ are complete. We assume the same alphabet.

Construct $A = (Q, \Sigma, q_0, F, \delta)$ as follows.

- $Q = Q_R \times Q_S$.
- $q_0 = (q_{0R}, q_{0S})$.
- $F = \{(q_r, q_s) \mid q_r \in F_R \textbf{ or } q_s \in F_S\}$.
- $\delta((q_r, q_s), a) = (q'_r, q'_s)$ where $\delta_R(q_r, a) = q'_r$ and $\delta_S(q_s, a) = q'_s$.

**Theorem 1.** $L(A) = R \cup S$.

Similarly, the same kind of construction shows that the intersection of any two regular stringsets is regular. Construct $B = (Q, \Sigma, q_0, F, \delta)$ as follows.

- $Q = Q_R \times Q_S$.
- $q_0 = (q_{0R}, q_{0S})$.
- $F = \{(q_r, q_s) \mid q_r \in F_R$ **and** $q_s \in F_S\}$.
- $\delta((q_r, q_s), a) = (q_r', q_s')$ where $\delta_R(q_r, a) = q_r'$ and $\delta_S(q_s, a) = q_s'$.

**Theorem 2.** $L(B) = R \cap S$.

Here are some additional questions we are interested in for regular stringsets R and S.

1. Is the complement of $R$ (denoted $\overline{R}$) a regular stringset?
2. Is $R \backslash S$ a regular stringset?
3. Can we decide whether $R \subseteq S$?
4. Is $RS$ a regular stringset? (Note $RS = \{rs \mid r \in R$ and $s \in S\}$
5. Is $R^*$ a regular stringset? (Note $R^0 = \{\lambda\}$, $R^n = R^{n-1}R$, $R^* = \bigcup_{n \in \mathbb{N}^0} R^n$)

The answers to all of these questions is Yes. With a little thought about complete DFA, the answers to first three follow very easily.

**Theorem 3.** *If $R$ is a regular stringset then the complement of $R$ is regular.*

**Proof** (Sketch). If $R$ is a regular stringset then there is a complete DFA $A = (Q, \Sigma, q_0, F, \delta)$ which recognizes it. Let $B = (Q, \Sigma, q_0, F', \delta)$ where $F' = Q \backslash F$. We claim $L(B) = \overline{R}$. □

**Corollary 1.** *If $R, S$ are regular stringsets then so is $R \backslash S$ since $R \backslash S = R \cap \overline{S}$.*

**Corollary 2.** *If $R, S$ are regular stringsets then it is decidable whether $R \subseteq S$ since $R \subseteq S$ iff $R \backslash S = \varnothing$.*

**Corollary 3.** *If $R, S$ are regular stringsets then it is decidable if $R = S$ since $R = S$ iff $R \subseteq S$ and $S \subseteq R$.*

We postpone explaining how and why for the last two questions.

## 2.4 Nonregular stringsets

Regular stringsets were defined as those subsets of $\Sigma^*$ whose membership problem was solvable with a deterministic finite-state acceptor. It was also expressed that finite-state solvable membership problems invoke constant memory. This can be understood to mean that the number of states is constant and does not increase even as the words grow longer and longer.

In this section, we give a few examples of stringsets whose membership problem is not solvable with any DFSA. In each case, I hope to convey that in order to solve the membership problem for any word, the number of states needs to increase as words get longer.

**Example 5.** Let $\Sigma = \{a, b, c\}$.

1. The set of strings that with each string containing at least as many $a$s as $b$s. Formally, we can define this set of strings as follows. For each $a \in \Sigma$ and $w \in \Sigma^*$, let $|w|_a$ be the number of times $a$ occurs in $w$. So $|aabaa|_a = 4$ and $|caca|_c = 2$ and so on. Then the set of strings we are interested in can be expressed as $S_1 = \{w \mid |w|_a \geq |w|_b\}$.

2. The palindrome language. Recall that a palindrome is a word that is the same when written both forwards and backwards. So the palindrome language contains all and only those words that are palindromes. Formally, we can define this set of strings as follows. Let us write $w^R$ to express the reverse of $w$. So $abac^R = caba$. Then this stringset can be expressed as $S_2 = \left\{wxw^R \mid w \in \Sigma^*, x \in \{\lambda, a, b, c\}\right\}$.

3. The $a^nb^n$ stringset. This is pronounced "$a$ to the $n$, $b$ to the $n$". Recall that $a^n$ defines the string with $a$ concatenated to itself $n$ times. So $a^4 = aaaa$ and $c^3 = ccc$ and so on. Note $a^0 = \lambda$. So this stringset can be expressed as $S_3 = \{a^nb^n \mid 0 \leq n\}$.

### 2.4.1 Exercises

**Exercise 13.** For each of the above examples, we try to write a deterministic finite-state acceptor. This exercise will help us understand why it is impossible.

1. First write an acceptor for $S_1$ for words up to size 3. Next try to write the acceptor for words up to size 5. What is happening? What information is each state keeping tracking of?

2. First write an acceptor for $S_2$ for words up to size 2. Next try to write the acceptor for words up to size 4. What is happening? What information is each state keeping tracking of?

3. First write an acceptor for $S_3$ for words up to size 2. Next try to write the acceptor for words up to size 4. What about up to size 6? What is happening? What information is each state keeping tracking of?

In each case, the information that the states need to keep track of grows as words get longer. This is the basic insight into why the problem of these stringsets (and many like them) cannot be solved with finite-state acceptors.

### 2.4.2 Formal Analysis

Formal proofs that these are nonregular exist, based on abstract properies of regular stringsets as discussed in (Sipser, 1997; Hopcroft *et al.*, 2001) and elsewhere.

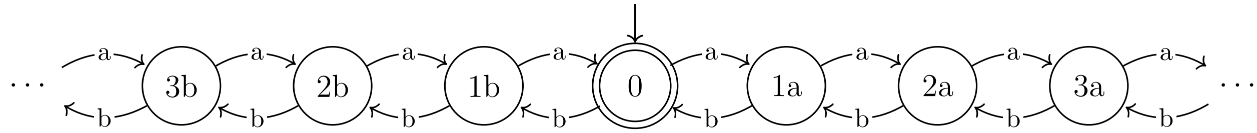
Here is one way to provide a rigorous proof with what we have learned so far.

Recall that when minimizing a DFSA $A$, we had to determine whether two states did the "same work" or not. We said states did different work, if there was a string that *distinguished* them. A string *distinguishes* state $q$ from $r$ if $A$ would transition to a final state from $q$ but to a non-final state from $r$ (or vice versa). We can use distinguishing strings to see that a DFSA for the above stringsets would have infinitely many states.

Suppose there was a DFSA $A$ for $S_1$. Let $q_0$ be the inital state of $A$ and let $q_a = \delta(q_0, a)$. Now we can ask are there any strings which distinguish $q$ from $q_a$? The answer is Yes. The string $b$ distinguishes them because $A$ must transition to an accepting state from $q_a$ with $b$, but must transition to a non-accepting state from $q_0$ with $b$. Thus $q_0$ and $q_a$ are distinct states.

We can repeat this reasoning. Let $q_{aa} = \delta(q_a, a)$. Is there a string which distinguishes $q_{aa}$ from $q_a$? Yes, in fact the string $bb$ distinguishes them because $A$ must transition to an accepting state from $q_{aa}$ with $bb$, but must transition to a non-accepting state from $q_a$ with $bb$. So $q_a$ and $q_{aa}$ are distinct states. What about $q_{aa}$ and $q_0$? They are distinct states too as witnessed by the distinguishing string $b$. So now we have three distinct states.

More generally, define state $q_i$ to be $\delta^*(q_0, a^i)$. For any numbers $n, m$ with $n > m$, one can show that $q_n$ is a distinct state from $q_m$. We have to find a distinguishing string for these two states. The string $b^n$ is such a string. The DFSA $A$ must transition to an accepting state from $q_n$ with $b^n$ because the word $a^n b^n$ has at least as many $a$s as $b$s. However, $A$ transitions to an non-accepting state from $q_m$ with $b^n$ because the word $a^m b^n$ has more $b$s as $a$s since $n > m$. Thus $A$ must have infinitely many states if it is to accept words of arbitrary length. And we haven't even looked at words beginning with $b$s yet!



**Exercise 14.** Present an argument like the one above for $S_2$ and $S_3$.

# Chapter 3

# Tree Acceptors

## 3.1 Deterministic Finite-state Bottom-up Tree Acceptors

### 3.1.1 Orientation

This section is about deterministic bottom-up finite-state tree acceptors. The term *finite-state* means that the memory is bounded by a constant, no matter the size of the input to the machine. The term *deterministic* means there is single course of action the machine follows to compute its output. The term *acceptor* means this machine solves *membership problem*: given a set of objects $X$ and input object $x$, does $x$ belong to $X$? The term *tree* means we are considering the membership problem over *treesets*. The term *bottom-up* means that for each node $a$ in a tree, the computation solves the problem by assigning states to the children of $a$ before assigning states to $a$ itself. This contrasts with *top-down* machines which assign states to $a$ first and then the children of $a$. Visually, these terms make sense provided the root of the tree is at the top and branches of the tree move downward.

*Acceptor* is synonymous with *recognizer*. *Treeset* is synonymous with *tree language*.

A definitive reference for finite-state automata for trees is freely available online. It is "Tree Automata Techniques and Applications" (TATA) (Comon *et al.*, 2007). The presentation here differs from the one there, as mentioned below.

### 3.1.2 Definitions

We will use the following definition of trees.

**Definition 15** (Trees). *We assume an alphabet $\Sigma$ and symbols [ ] not belonging to $\Sigma$.*

**Base Cases:** *For each $a \in \Sigma$, $a[\ ]$ is a tree.*

**Inductive Case:** *If $a \in \Sigma$ and $t_1 t_2 \dots t_n$ is a string of trees of length $n$ then $a[t_1 t_2 \cdot \dots t_n]$ is a tree.*

*Let $\Sigma^T$ denote the set of all trees of finite size using $\Sigma$. We also write $a[\lambda]$ for $a[\,]$.*

It will be helpful to review the following concepts related to functions: *domain, co-domain, image*, and *pre-image*. A function $f : X \rightarrow Y$ is said to have domain $X$ and co-domain $Y$. This means that if $f(x)$ is defined, we know $x \in X$ and $f(x) \in Y$. However, $f$ may not be defined for all $x \in X$. Also, $f$ may not be *onto* $Y$, there may be some elements in $Y$ that are never "reached" by $f$.

This is where the concepts *image* and *pre-image* come into play. The image of $f$ is the set $\{f(x) \in Y \mid x \in X, f(x) \text{ is defined}\}$. The pre-image of $f$ is the set $\{x \in X \mid f(x) \text{ is defined}\}$. So the pre-image of $f$ is the subset of the domain of $f$ where $f$ is defined. The image of $f$ is the corresponding subset of the co-domain of $f$.

With this in place, we can define our first tree acceptor.

**Definition 16** (DFBTA)**.** *A* Deterministic Bottom-up Finite-state Acceptor (DFBTA) *is a tuple $(Q, \Sigma_r, F, \delta)$ where*

- *$Q$ is a finite set of states;*
- *$\Sigma$ is a finite alphabet;*
- *$F \subseteq Q$ is a set of* accepting *(*final*) states; and*
- *$\delta : Q^* \times \Sigma \rightarrow Q$ is* the transition function. The pre-image of $\delta$ must be finite*. This means we can write it down—for example, as a list.*

We use the transition function $\delta$ to define a new function $\delta^* : \Sigma^T \rightarrow Q$ as follows.

$$\begin{aligned} \delta^*(a[\lambda]) &= \delta(\lambda, a) \\ \delta^*(a[t_1 \cdots t_n]) &= \delta\big(\delta^*(t_1) \cdots \delta^*(t_n), a\big) \end{aligned} \tag{3.1}$$

There are some important consequences to the formulation of $\delta^*$ shown here. One is that $\delta^*$ is undefined on tree $a[t_1 \cdots t_n]$ if no transition $\delta(q_1 \cdots q_n, a)$ is defined.

(Also, I am abusing notation since $\delta^*$ is strictly speaking not the transitive closure of $\delta$.)

**Definition 17** (Treeset of a DFBTA)**.** *Consider some DFBTA $A = (Q, \Sigma, F, \delta)$ and tree $t \in \Sigma^T$. If $\delta^*(t)$ is defined and belongs to $F$ then we say $A$* accepts/recognizes *$t$. Otherwise $A$* rejects *$t$. The treeset recognized by $A$ is $L(A) = \{t \in \Sigma^T \mid \delta^*(t) \in F\}$.*

The use of the 'L' denotes "Language" as treesets are traditionally referred to as *formal tree languages.*

We observe that since the pre-image of $\delta$ is finite, there is some $n$ such that for all $(\vec{q}, a, q') \in \delta$, it is the case that $|\vec{q}| \leq n$. In other words, $\delta$ is effectively a function from $Q^n \times \Sigma$ to $Q$. We say $\delta$ is $n$-wide and any DFBTA with an $n$-wide $\delta$ is also called $n$-wide.

**Definition 18** ($n$-wide Recognizable Treesets)**.** *A treeset is $n$-*wide recognizable *if there is a n-wide DFBTA that recognizes it.*

**Definition 19** (Recognizable Treesets)**.** *A treeset is* recognizable *if there exists $n \in \mathbb{N}$, such that an n-wide DFBTA that recognizes it.*

### 3.1.3 Notes on Definitions

We have departed a bit from standard definitions. In particular, most introductions to tree automata make use of a particular kind of alphabet called a *ranked alphabet.* A ranked alphabet $\Sigma_r$ is a finite alphabet $\Sigma$ with an arity function $ar : \Sigma \rightarrow \mathbb{N}$. We write $\Sigma_r = (\Sigma, ar)$. The idea is that each symbol comes pre-equipped with a number which indicates how many children it has in trees. This is reasonable provided a node's label determines how many children it can have.

Strictly speaking, a ranked alphabet is not a necessary feature of tree automata. There are two substantive reasons to adopt it. First, using it helps ensure that the transition function is finite. (So it can accomplish the same thing as our requirement that the pre-image of $\delta$ be finite.). Second, it can help ensure our transition function is total; that is, defined for every element of the alphabet and the possible states of its children.

A ranked alphabet effectively defines a maximum width $n$ for treees where $n = \mathtt{max}\{ar(a) \mid a \in \Sigma\}$. Thus a DFBTA defined with a ranked alphabet will always recognize a treeset which is a subset of $\Sigma^{\mathrm{T},n}$.

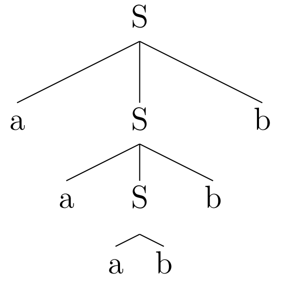
### 3.1.4 Examples

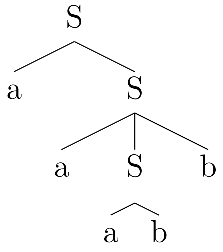
**Example 6.** Let $A = (Q, \Sigma, F, \delta)$ with its parts defined as follows.

- $Q = \{q_a, q_b, q_S\}$
- $\Sigma = \{a, b, S\}$
- $F = \{q_S\}$

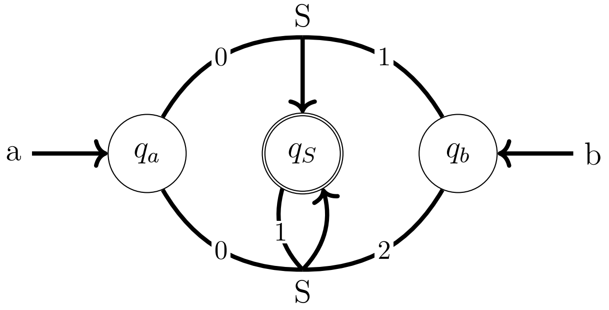
- $\delta(\lambda, a) = q_a$
- $\delta(\lambda, b) = q_b$
- $\delta(q_a q_b, S) = q_S$
- $\delta(q_a q_S q_b, S) = q_S$

Let us see how the acceptor $A$ processes the two trees below as inputs.





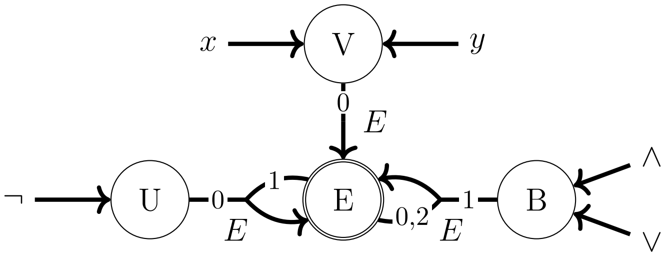
A Lambert graph of this automaton is shown below.



**Example 7.** The next example regards propositional logic. Propositional logic has one unary operator, negation (¬), and two binary operators: conjunction (∧) and disjunction (∨). We consider the case with two variables $\{x, y\}$. We can define trees of propositional logic as follows.

- x[] and y[] are trees of propositional logic.
- If A and B are trees of propositional logic, then so are ¬ [A], ∧[A,B], and ∨[B,A].

A Lambert graph of the automata recognizing this language is shown below.



### 3.1.5 Observations

- For every symbol $a \in \Sigma$ which can be leaf in a tree, you will need to define a transition $\delta(\lambda, a)$.
- For every symbol $a \in \Sigma$ which can have $n$ children, you will need to define a transition $\delta(q_1 \cdots q_n, a)$.

### 3.1.6 Connection to Context-Free Languages

Recognizable treesets are closely related to the derivation trees of context-free languages.

**Theorem 4.**

- *Let G be a context-free word grammar, then the set of derivation trees of L(G) is a recognizable tree language.*
- *Let L be a recognizable tree language then Yield(L) is a context-free word language.*

- *There exists a recognizable tree language not equal to the set of derivation trees of any context-free language. Thus the class of derivation treesets of context-free word languages is a proper subset of the class of recognizable treesets.*

## 3.2 Deterministic Top-down Finite-state Tree Acceptors

### 3.2.1 Orientation

This section is about deterministic top-down finite-state tree acceptors. The term *finite-state* means that the memory is bounded by a constant, no matter the size of the input to the machine. The term *deterministic* means there is single course of action the machine follows to compute its output. The term *acceptor* means this machine solves *membership problem*: given a set of objects $X$ and input object $x$, does $x$ belong to $X$? The term *tree* means we are considering the membership problem over *treesets.* The term *top-down* means that for each node $a$ in a tree, the computation solves the problem by assigning a state to the parent of $a$ before assigning a state to $a$ itself. This contrasts with *bottom-up* machines which assign states to the children of $a$ first and then $a$. Visually, these terms make sense provided the root of the tree is at the top and branches of the tree move downward.

*Acceptor* is synonymous with *recognizer*. *Treeset* is synonymous with *tree language.*

A definitive reference for finite-state automata for trees is freely available online. It is "Tree Automata Techniques and Applications" (TATA) (Comon *et al.*, 2007). The presentation here differs from the one there, as mentioned below.

### 3.2.2 Definition

**Definition 20** (DTFTA)**.** *A* Deterministic Top-down Finite-state Acceptor (DTFTA) *is a tuple* $(Q, \Sigma_r, F, \delta)$ *where*

- $Q$ *is a finite set of states;*
- $q_0$ *is a initial state;*
- $\Sigma$ *is a finite alphabet;*
- $\delta : Q \times \Sigma \times \mathbb{N} \to Q^*$ *is the transition function. Note the pre-image of* $\delta$ *is necessarily finite.*

The transition function takes a state, a letter, and a number $n$ and returns a string of states. The idea is that the length of this output string should be $n$. Basically, when moving top-down, the states of the child sub-trees depend on these three things: the state of the parent, the label of the parent, and the number of children the parent has.

We use the transition function $\delta$ to define a new function $\delta^* : Q \times \Sigma^T \to Q^*$ as follows.

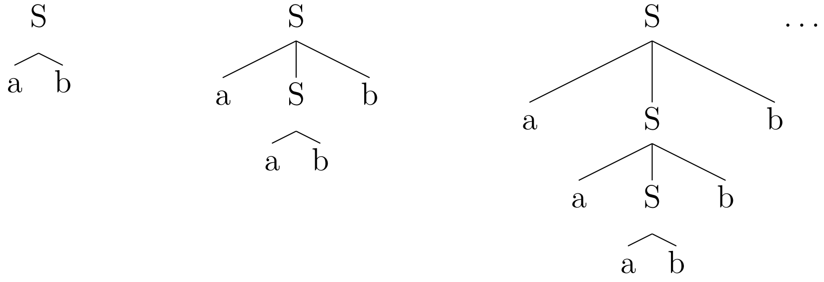
$$\begin{array}{rcl} \delta^*(q, a[\lambda]) & = & \delta(q, a, 0) \\ \delta^*(q, a[t_1 \cdots t_n]) & = & \delta^*(q_1, t_1) \cdots \delta^*(q_n, t_n) \text{ where } \delta(q, a, n) = q_1 \cdots q_n \end{array} \tag{3.2}$$

As before, there are some important consequences to the formulation of $\delta^*$. One is that $\delta^*$ is undefined on tree $a[t_1 \cdots t_n]$ if transition $\delta(q, a, n)$ does not return a string from $Q^*$ of length $n$. (Also, I am abusing notation since $\delta^*$ is strictly speaking not the transitive closure of $\delta$.)

**Definition 21** (Treeset of a DTFTA)**.** *Consider some DTFTA $A = (Q, \Sigma, F, \delta)$ and tree $t \in \Sigma^T$. If $\delta^*(q_0, t)$ is defined and equals $\lambda$ then we say $A$* accepts/recognizes *$t$. Otherwise $A$* rejects *$t$. Formally, the treeset recognized by $A$ is $L(A) = \{t \in \Sigma^T \mid \delta^*(t) = \lambda\}$.*

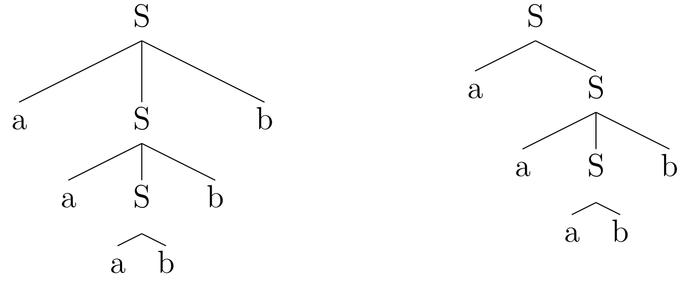
The use of the 'L' denotes "Language" as treesets are traditionally referred to as *formal tree languages.*

**Example 8.** Recall the example from last week which generates trees like



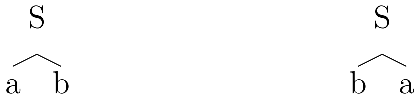
- $Q = \{q_a, q_b, q_S\}$
- $\Sigma = \{a, b, S\}$
- $q_0 = q_S$
- $\delta(q_a, a, 0) = \lambda$
- $\delta(q_b, b, 0) = \lambda$
- $\delta(q_S, S, 3) = q_a q_S q_b$
- $\delta(q_S, S, 2) = q_a q_b$

Let us see how the acceptor $A$ processes the two trees below as inputs.



**Theorem 5.** *Every treeset recognizable by a DTFTA is recognizable, but there are recognizable treesets which cannot be recognized by a DTFTA.*

The following example helps show why this is the case. Consider the treeset $T$ containing only the two trees shown below.

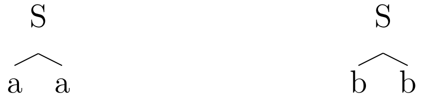


This is a recoginiable treeset because the DBFTA below recognizes exactly these two trees and no others.

- $Q = \{q, q_S\}$
- $\Sigma = \{a, b, S\}$
- $q_0 = q_S$

- $\delta(\lambda, a) = q_a$
- $\delta(\lambda, b) = q_b$
- $\delta(q_a q_b, S) = q_S$
- $\delta(q_b q_a, S) = q_S$

Notice that this DBFTA fails on these two trees.



A DTFTA cannot recognize the trees in $T$ without also recognizing the trees shown immediately above. This is because moving top down there can only be one value for $\delta(q_S, S, 2)$. Suppose it equals $q_1 q_2$. To recognize the first tree, we would also have to makes sure that $\delta(q_1, a, 0)$ and $\delta(q_2, b, 0)$ are defined. Similarly, to recognize the second tree, we would have to makes sure that $\delta(q_1, b, 0)$ and $\delta(q_2, a, 0)$ are defined. But it follows then that the aforementioned trees above are also recognized by this DTFTA. For instance the tree with two $a$ leaves is recognized because both $\delta(q_1, a, 0)$ and $\delta(q_2, a, 0)$ are defined. Thus no DTFTA recognizes $T$.

### 3.2.3 Observations

- For every symbol $a \in \Sigma$ which can be leaf in a tree, you will need to define a transition $\delta(q, a, 0) = \lambda$.

- For every symbol $a \in \Sigma$ which can have $n$ children, you will need to define a transition $\delta(q, a, n) = q_1 \cdots q_n$.

## 3.3 Properties of recognizable tree languages

**Theorem 6** (Closure under Boolean operations)**.** *The class of recognizable tree languages are closed under union and intersection.*

*The class of $n$-wide recognizable tree languages are closed under union, intersection, and complementation with respect to the $\Sigma^{\mathrm{T},n}$.*

The proofs of these cases are very similar to the ones for finite-state acceptors over strings. For every DBFTA $A$ recognizing a treeset $T$, it can be made *complete* by adding a sink state and transitions to it. Then product constructions can be used to establish closure under union and intersection. Closure under complement is established the same as before too: everything is the same except the final states are now the non-final states of $A$.

**Theorem 7** (Minimal, determinstic, canonical form)**.** *For every recognizable tree language $T$, there is a unique, smallest DBFTA $A$ which recognizes $T$. That is, if DBFTA $A'$ also recognizes $T$ then there at least as many states in $A'$ as there are in $A$.*

## 3.4 Connection to Context-Free Languages

Recognizable treesets are closely related to the derivation trees of context-free languages.

**Theorem 8.**

- *Let $G$ be a context-free word grammar, then the set of derivation trees $D_T(G)$ is a recognizable tree language.*
- *There exists a recognizable tree language not equal to the set of derivation trees of any context-free language. Thus the class of derivation treesets of context-free word languages is a proper subset of the class of recognizable treesets.*
- *Let $L$ be a recognizable tree language then* $\texttt{yield}(L)$ *is a context-free word language.*

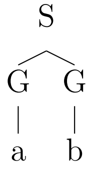
Each of these has a straightforward explanation.

For (1), the recognizable tree language which recognizes $D_T(G)$ for a CFG $G$ can be constructed based on the rules of $G$. For each symbol $a$ in $N$, the DBFTA should include $\delta(\lambda, a) = q_a$. And, for each rule $A \rightarrow B_1 \cdots B_n$ in $R$, the DBFTA should include $\delta(q_{B_1} \cdots q_{B_n}, A) = q_A$. That's it!

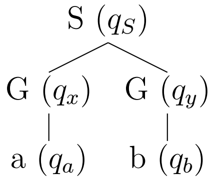
For (2), consider the DBFTA $A$ shown below. The claim is that there is no CFG whose derivation language is exactly this recognizable treeset. The only tree in $L(A)$ is (t1), which is shown below $A$ at left. Tree (t2) shows (t1) with the states $A$ assigns to its subtrees.

$$(A) \qquad \begin{array}{rcl} Q &=& \{q_S, q_a, q_b, q_x, q_y\} \\ \Sigma &=& \{S, G, a, b\} \\ F &=& \{q_S\} \\ \delta(q_x q_y, S) &=& q_S \end{array} \qquad \begin{array}{rcl} \delta(\lambda, a) &=& q_a \\ \delta(\lambda, b) &=& q_b \\ \delta(q_a, G) &=& q_x \\ \delta(q_b, G) &=& q_y \end{array}$$

(t1)

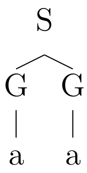


(t2)

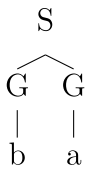


That no CFG can recognize this treeset follows from the fact that such any CFG $G$ which includes the tree above will need to have the following rules: $S \rightarrow GG, G \rightarrow a, G \rightarrow b$. But then this $G$ will not only generate (t1) above but also the derivation trees shown below.

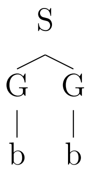
(t3)



(t4)



(t5)



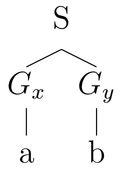
Thus $L(A) \neq D_T(G)$.

This example shows that the states of the DBFTA are more abstract than the labels on the nodes. The reason recognizable tree languages are more expressive than the derivation treesets of CFGs follows from this. The DBFTA uses states $q_x$ and $q_y$ to distinguish the subtrees bearing the label $G$. But the CFG cannot distinguish these trees in this way.

For (3), observe that we can write a CFG that generates the same stringset as the one above. For instance, we could write a CFG with the rule $S \rightarrow ab$.

More generally, though, we can always write a CFG that puts the state information into the nodes themselves. The trees in the derivation treeset for this CFG would be "structurally the same" as the trees in the recognizable treeset, but the labels on the nodes would be different. So they are not the same trees. In the example above for instance we can write a CFG with rules $S \rightarrow G_x G_y, G_x \rightarrow a, G_x \rightarrow b$. There is only one tree in this CFG's derivation treeset shown below.

(t6)



Importantly, (t6) is not the same as (t1). The inner nodes are labeled differently! So they are different trees. However, they only differ with respect to how the inner nodes are labeled, so it follows that the stringsets obtained by taking the `yield` of these trees are the same. This is the kind of argument used to show that the yield of any recognizable treeset is a context-free language.

# Chapter 4

# String Transducers

## 4.1 Deterministic Finite-state String Transducers

### 4.1.1 Orientation

This section is about deterministic finite-state transducers for strings. The term *finite-state* means that the memory is bounded by a constant, no matter the size of the input to the machine. The term *deterministic* means there is single course of action the machine follows to compute the output from some input. The term *transducer* means this machine solves *transformation problems*: given an input object $x$, what object $y$ is $x$ transformed into? The term *string* means we are considering the transformation problem from strings to objects. So $x$ is a string. As we will see, we can easily write transducers where $y$ is a string, natural number, real number, or even a finite stringset! We will also see that DFSAs are a specific case of DFSTs.

However, we first define the output of the transformation to be a string. Then we will generalize it.

### 4.1.2 Definitions

**Definition 22.** *A* deterministic finite-state string-to-string transducer (DFST) *is a tuple* $T = (Q, \Sigma, \Delta, q_0, v_0, \delta, F)$ *where*

- $Q$ *is a finite set of states;*
- $\Sigma$ *is a finite set of symbols (*the input alphabet*);*
- $\Delta$ *is a finite set of ouput symbols (*the output alphabet*);*
- $q_0 \in Q$ *is the* initial *state;*
- $v_0 \in \Delta^*$ *is the* initial string*;*
- $\delta$ *is a function with domain* $Q \times \Sigma$ *and co-domain* $Q \times \Delta^*$*. It is called the* transition function*. If transition* $(q, a, r, v) \in \delta$ *it means that there is a transition from state* $q$ *to state* $r$ *reading letter* $a$ *and writing string* $v$*. It will be helpful to refer to the "first" and*

*"second" outputs of delta with $\delta_1$ and $\delta_2$ respectively. So for all $(q, a, r, v) \in \delta$, we have $\delta_1(q, a) = r$ and $\delta_2(q, a) = v$. These are are the "state" transition and the "output" transitions, respectively.*

- *$F$ is a function with domain $Q$ and co-domain $\Delta^*$. Let's call it the* final function*.*

For each transducer $T$, we can define a new function "process" $\pi : Q \times \Delta^* \times \Sigma^* \to \Delta^*$ as follows. Ultimately, $\pi(q, v, w)$ processes an input string $w$ letter by letter from a given state $q$ with a given output string $v$ and returns an output string.

$$\begin{array}{rcl} \pi(q, v, \lambda) & = & v \cdot F(q) \\ \pi(q, v, aw) & = & \pi\big((\delta_1(q, a),\ v \cdot \delta_2(q, a),\ w\big) \end{array} \tag{4.1}$$

Consider some DFST $T = (Q, \Sigma, \Delta, q_0, v_0, \delta, F)$ and string $u \in \Sigma^*$. Then $T(u) = pi(q_0, v_0, u)$. We say $T$ *transforms* $u$ into $v$.

**Definition 23.** *The function defined by $T$ is $\big\{(u, v) \in \Sigma^* \times \Delta^* \mid \pi(q_0, v_0, u) = v\big\}$.*

We write $T(u) = v$ iff $(u, v) \in T$.

**Definition 24.** *A string-to-string function is called* sequential *if there is a DFST that recognizes it.*

(Sequential functions are also often called subsequential functions. The nomenclature is unfortunate and different people have different opinions about it.) The important thing is that they are *deterministic* on the input and one should always check the definitions and not rely on names.

### 4.1.3 Exercises

**Exercise 15.** Let $\Sigma = \Delta = \{a, e, i, o, u, p, t, k, b, d, g, m, n, s, z, l, r\}$.

1. Write a transducer that prefixes *pa* to all words.
2. Write a transducer that suffixes *ing* to all words.
3. Write a transducer that deletes word initial vowels.
4. Write a transducer that voices obstruents which occur immediately after nasals.
5. Write a transducer that deletes word final vowels. So $T(abba) = abb$ and $T(pie) = pi$.
6. Write a transducer that voices obstruents intervocalically.

Note that the transition function and the final function can be partial functions. In this case, the transducer is *incomplete* in the sense it is not defined for all inputs. As before, we will strive to make our sequential transducers describe total functions.

## 4.2 Some Closure Properties of Sequential functions

**Theorem 9** (Closure under composition)**.** *If $f, g$ are sequential functions then so is $f \circ g$.*

**Theorem 10** (Closure under intersection)**.** *If $f, g$ are sequential functions then so is $f \cap g$.*

**Theorem 11** (Minimal canonical form)**.** *For every sequential string-to-string function $f$, it is possible to compute a DFST $T$ such that $T$ is equivalent to $f$ and no other DFST $T'$ equivalent to $f$ has fewer states than $T$.*

The closure theorems follow from proofs which use the product construction. The minimal cananoical form result is due to Choffrut (see his 2003 survey).

Sequential functions are not closed under union. This is because they are functions and so each input has a unique output.

## 4.3 Generalizing sequential functions with monoids

A *monoid* is a mathematical term which means any set which is closed under some associative binary operation with an identity. So if $(S, *, 1)$ is a monoid then for all $x, y \in S$:

1. is the case that $x * y$ is in $S$ too (Closure under $*$)
2. $(x * y) * z = x * (y * z)$ (Associativity)
3. $1 * x = x * 1 = x$ (1 is the identity)

It is typical to refer to $*$ as "times", "multiplication" or as a product. It is also typical to refer to 1 as the "identity", "unit" or "one".

$\Sigma^*$ is closed under the binary operation of concatenation. Also the empty string behaves like the identity with respect to concatenation. So $(\Sigma^*, \ \cdot\ , \lambda)$ is a monoid. As we processed the input string, we moved from state to state and updated the ouput value by concatenating strings along the output transitions. We can do the same thing and update the output value using some other product from another monoid.

**Example 9.** Here are some examples.

1. $(\{\texttt{True}, \texttt{False}\},\ \wedge,\ \texttt{True})$. Boolean values and conjunction. This monoid shows the membership problem is a special case of the transformation problem.
2. $(\mathbb{N},\ +,\ 0)$. Natural numbers and addition. Useful for counting!
3. $([0, 1],\ \times,\ 1)$. The real unit interval and multiplication. Useful for probabilities!
4. $(\mathbb{R},\ \times,\ 1)$. All real numbers and multiplication.
5. (FIN, $\cdot$, $\{\lambda\}$) where FIN is the class of finite stringsets and $(\cdot)$ is concatenation of stringsets.

   - FIN $= \{S \mid \exists n \in \mathbb{N} \text{ with } |S| = n\}$

- $S_1 \cdot S_2 = \{u \cdot v \mid u \in S_1, v \in S_2\}$.

6. There are many others!

This means we can generalize DFSTs to transducers which output elements from any monoid.

**Definition 25.** *A* generalized sequential transducer (GST) *is a tuple* $T = (Q, \Sigma, M, q_0, v_0, \delta, F)$ *where*

- $Q$ *is a finite set of states;*
- $\Sigma$ *is a finite set of symbols (*the input alphabet*);*
- $(M, *, 1)$ *is a monoid*
- $q_0 \in Q$ *is the* initial *state;*
- $v_0 \in M$ *is the* initial *value;*
- $\delta$ *is a function with domain* $Q \times \Sigma$ *and co-domain* $Q \times \mu$. *It is called the* transition function. *As before, from this we derive* $\delta_1 : Q \times \Sigma \rightarrow Q$ *and* $\delta_2 : Q \times \Sigma \rightarrow \mu$ *to be the state and output transition functions.*
- $F$ *is a function with domain* $Q$ *and co-domain* $M$. *Let's call it the* final function.

The process function $\pi$ is the same except we replace concatenation of the outputs with the monoid operator $*$.

$$\begin{aligned} \pi(q, v, \lambda) &= v * F(q) \\ \pi(q, v, aw) &= \pi\big((\delta_1(q, a),\ v * \delta_2(q, a),\ w\big) \end{aligned} \tag{4.2}$$

Then, the definition of the function computed by the transducer is identical to what was written formerly.

That's it!

### 4.3.1 Exercises

**Exercise 16.** Let $\Sigma = \{a, e, i, o, u, p, t, k, b, d, g, m, n, s, z, l, r\}$.

1. Write a transducer that counts how many NC (nasal-consonant) sequences occurs in the input word.
2. Write a transducer that optionally voices obstruents which occur immediately after nasals. So for in input like *anta* the output should be the set {*anda, anta*}. (Hint: use the FIN monoid).

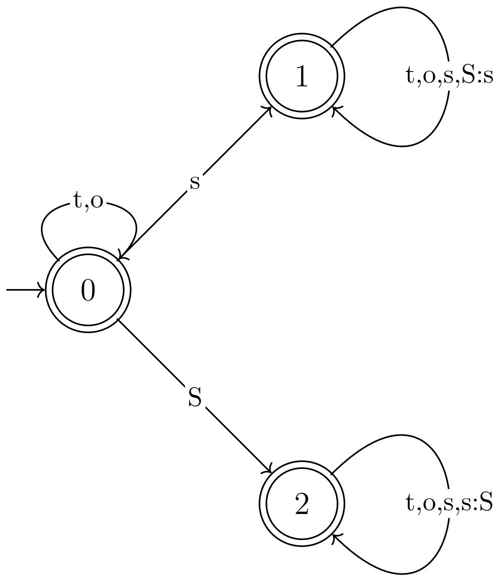
## 4.4 Learning more

Unfortunately, the material on deterministic transducers has yet to make its way into standard textbooks. Standard textbooks in computer science discuss non-deterministic finite-state transducers if they discuss transducers at all. Within computational linguistics where transducers are widely used, non-determinstic ones are the norm. See, for instance Roche and Schabes (1997); Beesley and Kartunnen (2003); Roark and Sproat (2007) and Jurafsky and Martin (2008). A notable exception is work by Mehyar Mohri (1997; 2005). The most textbook-like discussion of sequential functions I am aware of comes from Lothaire (2005, chapter 1).

My interest in sequential transducers stems from three interrelated facts. First, they appear sufficient to decribe morpho-phonologial generalizations in natural language (Jardine, 2016) and subsequent discussions. So the extra power that comes with non-deteminsitic transducers appears unnecessary in this domain. Second, sequential transducers have canonical forms (Choffrut's theorem), but non-deterministic ones do not. Third, the class of sequential transducers can be learned from examples, unlike the class of non-deterministic transducers (de la Higuera, 2010, chapter 18).

## 4.5 Left and Right Sequential Transducers

Below is a sequential transducer for progressive sibilant harmony. This means later sibilants agree in anteriority with the first sibilant in the word. The alphabet here is simply {s,S,t,o} with {s,S} signifying the two classes of sibilants and {t} other consonants, and {o} the vowels. We assume the initial value is $\lambda$ and for all $q$ the final function maps $q$ to $\lambda$. In the diagram, $a : b$ means $a$ is read as input and $b$ is ouput. If there is no colon and only $a$, then it means $a$ is read as input and $a$ is written as output.



Many examples of sibilant harmony in natural language are not progressive. They are in fact regressive. For instance here are some wonderfully long words in Samala (Chumash) (Applegate 1972). The underlying form is on the left and the surface form is on the right. There are some alternations in the vowels which we ignore here.

| | | |
|---|---|---|
| /ha-s-xintila-waʃ/ | [haʃxintilawaʃ] | ‘his former Indian name’ |
| /k-su-kili-mekeken-ʃ/ | [kʃuk’ilimeket͡ʃ] | ‘I straigten myself up’ |
| /k-su-al-puj-ʊn-ʃaʃi/ | [kʃalpujat͡ʃɨʃi] | ‘I get myself wet’ |
| /s-taja-nowon-waʃ/ | [ʃtojowonowaʃ] | ‘it stood upright’ |

**Exercise 17.** Explain why regressive sibilant harmony *cannot* be modeled with sequential transducers.

There is a straightforward way to address this issue. The transducers we are using process strings from left-to-right. However, transducers could also process strings from right-to-left. If we use the transducer above to process the Samala strings from right to left, we can model regressive sibilant harmony. The example below shows /stototooS/↦[StototooS].

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | s | | t | | o | | t | | o | | t | | o | | o | | S | | |
| ← | 3 | ← | 3 | ← | 3 | ← | 3 | ← | 3 | ← | 3 | ← | 3 | ← | 3 | ← | 3 | ← | 1 | ← |
| | | S | | t | | o | | t | | o | | t | | o | | o | | S | | |

If a transducer processes the string left-to-right, it is called a *left sequential* transducer. If it processes it right-to-left it is called a *right sequential* transducer.

**Theorem 12.** *Left sequential functions and right sequential functions are incomparable; that is, there are functions that are both left and right sequential; neither left nor right sequential; left but not right sequential; and right but not left sequential.*

Progressive sibilant harmony is a case in point. It is left sequential but not right sequential. On the other hand, regressive sibilant harmony is right sequential, but not left sequential. The identity function is both left and right sequential. Can you think of a a function which is neither left nor right sequential?

The recursive data structure we are using for strings is inherently left-to-right because the outermost element in the list structure is on the left. If we were to define lists so that the outermost element of the list structure was on the right, it would become natural to process strings right-to-left.

The Haskell implementation of strings is inherently left-to-right because the outermost element in the list structure is on the left. If we were to define lists so that the outermost element of the list structure was on the right, it would become natural to process strings right-to-left.

An easy way to simulate a right sequential transducer with the lists we have in Haskell is to do the following.

1. Implement left sequential transducers as we have done.
2. Before processing a string $w$, reverse it.
3. Then reverse the output of the transducer.

In other words, `transduce t w` will process the string left-to-right. However, the function `reverse (transduce t (reverse w))` will simulate $t$ processing $w$ right-to-left.

# Chapter 5

# Tree Transducers

## 5.1 Deterministic Bottom-up Finite-state Tree Transducers

### 5.1.1 Orientation

This section is about deterministic bottom-up finite-state tree transducers. The term *finite-state* means that the amount of memory needed in the course of computation is independent of the size of the input. The term *deterministic* means there is single course of action the machine follows to compute its output. The term *transducer* means this machine solves *transformation problem*: given an input object $x$, what object $y$ is $x$ transformed into? The term *tree* means we are considering the transformation problem from trees to trees. The term *bottom-up* means that for each node $a$ in a tree, the computation transforms the children of a node before transforming the node. This contrasts with *top-down* transducers which transform the children after transforming their parent. Visually, these terms make sense provided the root of the tree is at the top and branches of the tree move downward.

A definitive reference for finite-state automata for trees is freely available online. It is "Tree Automata Techniques and Applications" (TATA) (Comon *et al.*, 2007). The presentation here differs from the one there, as mentionepd below.

### 5.1.2 Definitions

Recall the definition of trees with a finite alphabet $\Sigma$ and the symbols [ ] not in $\Sigma$. All such trees belonged to the treeset $\Sigma^T$. In addition to this, we will need to define a new kind of tree which has *variables* in the leaves. I will call these trees *Variably-Leafed.* We assume a countable set of variables $X$ containing variables $x_1, x_2, \ldots$.

**Definition 26** (Variably-Leafed Trees)**.**

**Base Cases ($\Sigma$):** *For each $a \in \Sigma$, $a[\,]$ is a tree.*
**Base Cases ($X$):** *For each $x \in X$, $x[\,]$ is a tree.*

**Inductive Case:** *If $a \in \Sigma$ and $t_1 t_2 \ldots t_n$ is a string of trees of length $n$ then $a[t_1 t_2 \cdot \ldots t_n]$ is a tree.*

*Let $\Sigma^T[X]$ denote the set of all variably-leafed trees of finite size using $\Sigma$ and $X$.*

Note that $\Sigma^T \subsetneq \Sigma^T[X]$. In the tree transducers we write below, the variably-leafed trees will play a role in the omega function as well as the the intermediate stages of the transformation.

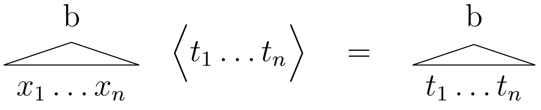
With this definition in place, we can define our first tree transducer.

**Definition 27** (DBFTA)**.** *A* Deterministic Bottom-up Finite-state Acceptor (DBFTT) *is a tuple $(Q, \Sigma, F, \delta)$ where*

- *$Q$ is a finite set of states;*
- *$\Sigma$ is a finite alphabet;*
- *$F \subseteq Q$ is a set of* accepting *(*final*) states; and*
- *$\delta : Q^* \times \Sigma \to Q \times \Sigma^T[X]$ is* the transition function. The pre-image of $\delta$ must be finite. *This means we can write it down—for example, as a list.*

  *If transition $(\vec{q}, a, r, t) \in \delta$ it means that if the children of node $a$ have been assigned states $\vec{q}$ then the state $r$ will be assigned to a the tree $t$ will be the output employed at this stage in the derivation. It will be helpful to refer to the "first" and "second" outputs of delta with $\delta_1$ and $\delta_2$ respectively. So for all $(\vec{q}, a, r, v) \in \delta$, we have $\delta_1(\vec{q}, a) = r$ and $\delta_2(\vec{q}, a) = t$. These are are the "state" transition and the "output" transitions, respectively.*

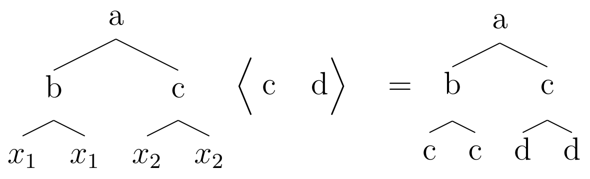
The key to understanding how a tree-to-tree transducer is defined is to understand how variables are used to rewrite and expand output trees. If $t_1 \ldots t_n$ is a list of trees and $t_x \in \Sigma^T[X]$ is a variably leafed tree with variables $x_1, \ldots x_n$ then let $t_x\langle t_1 \ldots t_n\rangle = t \in \Sigma^T$ obtained by replacing each variable $x_i$ in $t_x$ with $t_i$. Here is a visualization of this notation.



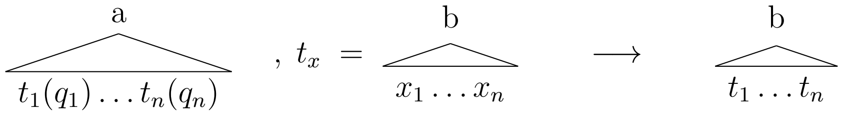
**Example 10.**

$$\Big[\ b[x_1\ x_1]\ \ c[x_2\ x_2]\ \Big]\Big\langle t_1 = c[\,], t_2 = d[\,]\Big\rangle = \Big[\ b[c[\,]\ c[\,]]\ \ c[d[\,]\ d[\,]]\ \Big]$$

Visually, we are taking the tree shown below and substituting $t_1$ for $x_1$ and $t_2$ for $x_2$.



Now we explain how substitution is used in the process of transducing a tree into another. Given $\delta_2(q_1 q_2 \ldots q_n, a) = t_x$ and a tree $a[t_1 t_2 \ldots t_n]$ with states $q_1 q_2 \ldots q_n$, respectively, then the output tree will be the one obtained by replacing each $x_i$ with $t_i$ in $t_x$. A schematic of this is shown below.



Then we can define a function "process" $\pi : \Sigma^T \to Q \times \Sigma^T$ which will process the tree and produce its output. It is defined recursively as follows.

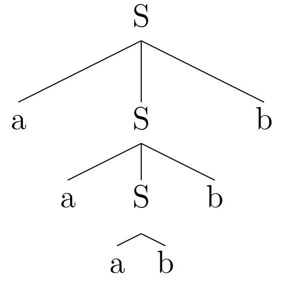
$$\begin{aligned} \pi(a[\,]) &= \big(\delta_1(\lambda, a), \delta_2(\lambda, a)\big) \\ \pi(a[t_1 \cdots t_n]) &= (q, t) \text{ where} \\ & \qquad q = \delta_1\big(q_1 \cdots q_n, a\big) \text{ and} \\ & \qquad t = \delta_2\big(q_1 \cdots q_n, a\big)\langle s_1 \cdots s_n \rangle \text{ and} \\ & \qquad (q_1, s_1) \cdots (q_n, s_n) = \pi(t_1) \cdots \pi(t_n) \end{aligned} \tag{5.1}$$

**Definition 28** (Tree-to-tree function of a DBFTT)**.** *The function defined by the transducer $T$ is $\big\{(t, s) \mid t, s \in \Sigma^T, \pi(t) = (q, s), q \in F\big\}$. If $(t, s)$ belongs to this set, we say $T$* transduces *$t$ to $s$ and write $T(t) = s$.*

**Example 11.** Consider the transducer $T$ constructed as follows.

- $Q = \{q_a, q_b, q_S\}$
- $\Sigma = \{a, b, S\}$
- $F = \{q_S\}$
- $\delta_1(\lambda, a) = q_a$
- $\delta_1(\lambda, b) = q_b$
- $\delta_1(q_a q_b, S) = q_S$
- $\delta_1(q_a q_S q_b, S) = q_S$
- $\delta_2(\lambda, a) = a[\,]$
- $\delta_2(\lambda, b) = b[\,]$
- $\delta_2(q_a q_b, S) = S[x_2 x_1]$
- $\delta_2(q_a q_S q_b, S) = S[x_3 x_2 x_1]$

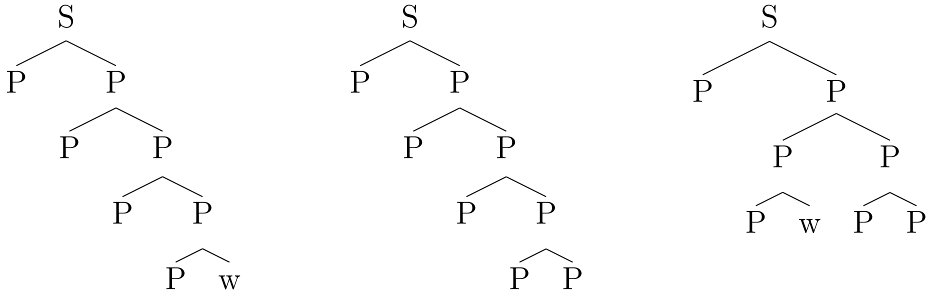
Let us work out what $T$ transforms the tree below into.



**Example 12.** Consider the transducer $T$ constructed as follows. We let $Q = \{q_w, q_p, q_S\}$, $\Sigma = \{w, P, S\}$, and $F = \{q_S\}$. The transition and ouput functions are given below.

- $\delta(\lambda, P) = q_p, P[\,]$
- $\delta(\lambda, w) = q_w, w[\,]$
- $\delta(q_pq_p, P) = q_p, P[x_1x_2]$
- $\delta(q_wq_p, P) = q_w, P[x_1x_2]$
- $\delta(q_pq_w, P) = q_w, P[x_1x_2]$
- $\delta(q_pq_w, S) = q_S, S[\ w[\,]\ S[x_1x_2]\ ]$
- $\delta(q_wq_p, S) = q_S, S[\ w[\,]\ S[x_1x_2]\ ]$
- $\delta(q_pq_p, S) = q_S, S[x_1x_2]$

Let us work out how $T$ transforms the trees below.



## 5.2 Deterministic Top-down Finite-state Tree Transducers

### 5.2.1 Orientation

This section is about deterministic bottom-up finite-state tree transducers. The term *finite-state* means that the amount of memory needed in the course of computation is independent of the size of the input. The term *deterministic* means there is single course of action the machine follows to compute its output. The term *transducer* means this machine solves *transformation problem*: given an input object $x$, what object $y$ is $x$ transformed into? The term *tree* means we are considering the transformation problem from trees to trees. The term *top-down* means that for each node $a$ in a tree, the computation transforms the node before transforming its children. This contrasts with *bottom-up* transducers which transform the children before transforming their parent. Visually, these terms make sense provided the root of the tree is at the top and branches of the tree move downward.

A definitive reference for finite-state automata for trees is freely available online. It is "Tree Automata Techniques and Applications" (TATA) (Comon *et al.*, 2007). The presentation here differs from the one there, as mentioned below.

### 5.2.2 Definitions

As before, we use variably leafed trees $\Sigma^T[X]$.

**Definition 29** (DTFTT)**.** *A* Deterministic Top-down Finite-state Acceptor (DTFTT) *is a tuple* $(Q, \Sigma_r, q_0, \delta)$ *where*

- $Q$ *is a finite set of states;*
- $\Sigma$ *is a finite alphabet;*
- $q_0 \in Q$ *is the initial state; and*
- $\delta : Q \times \Sigma \times \mathbb{N} \to Q^* \times \Sigma^T[X]$ *is* the transition function. *As before, we will derive "state" and "output" transitions, and notate them with* $\delta_1$ *and* $\delta_2$*, respectively. So for all* $(q, a, \vec{r}, t) \in \delta$*, we have* $\delta_1(q, a) = \vec{r}$ *and* $\delta_2(q, a) = t$*.*

We also define the "process" function $\pi : Q \times \Sigma^T \to \Sigma^T$ which will process the tree and produce its output. It is defined as follows.

$$\begin{aligned} \pi(q, a[\,]) &= \delta_2(q, a, 0) \\ \pi(q, a[t_1 \cdots t_n]) &= \delta_2(q, a, n)\big\langle \pi(q_1, t_1) \cdots \pi(q_n, t_n)\big\rangle \\ &\quad \text{where } q_1 \cdots q_n = \delta_1(q, a, n) \end{aligned} \tag{5.2}$$

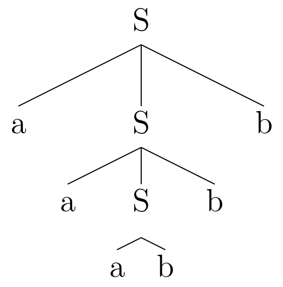
Intuitively, $\delta_2$ transforms the root node into a variably leafed tree. The variables are replaced with the children of the root node. These children are also trees with states assigned by $\delta_1$. Then $\pi$ transforms each tree-child as well.

**Definition 30** (Tree-to-tree function of a DTFTT)**.** *The function defined by the transducer* $T$ *is* $\big\{(t, s) \mid t, s \in \Sigma^T, \pi(q_0, t) = s\big\}$*. If* $(t, s)$ *belongs to this set, we say* $M$ transduces $t$ *to* $s$ *and write* $T(t) = s$*.*

**Example 13.** Consider the transducer $T$ constructed as follows.

- $Q = \{q, q_S\}$
- $\Sigma = \{a, b, S\}$
- $q_0 = q_S$
- $\delta_1(q, a, 0) = \lambda$
- $\delta_1(q, b, 0) = \lambda$
- $\delta_1(q_S, S, 3) = q q_S q$
- $\delta_1(q_S, S, 2) = qq$
- $\delta_2(q, a, 0) = a[\,]$
- $\delta_2(q, b, 0) = b[\,]$
- $\delta_2(q, S, 3) = S[x_3 x_2 x_1]$
- $\delta_2(q, S, 2) = S[x_2 x_1]$

Let see how $T$ transforms the tree below.



**Exercise 18.** Recall the "wh-movement" example from before. Explain why this transformation *cannot* be computed by a deterministic top-down tree transducer.

## 5.3 Theorems about Deterministic Tree Transducers

**Theorem 13** (composition closure)**.** *The class of deterministic bottom-up transductions is closed under composition, but the class of top-down deterministic transductions is not.*

**Theorem 14** (Incomparable)**.** *The class of deterministic bottom-up transductions is incomparable with the the class of top-down deterministic transductions.*

This theorem is based on the same kind of examples which separated the left and right sequential functions. Let relations $U = \{(f^n a, f^n a) \mid n \in \mathbb{N}\} \cup \{(f^n b, g^n b) \mid n \in \mathbb{N}\}$ and $D = \{(ff^n a, ff^n a) \mid n \in \mathbb{N}\} \cup \{(gf^n a, gf^n b) \mid n \in \mathbb{N}\}$. $U$ is recognized by a DBFTT but not any DTFTT and $D$ is recognized by a DTFTT but not any DBFTT.

# Chapter 6

# Nondeterminism

## 6.1 Non-Determinism

So far all of the finite-state automata we have considered have been deterministic. Informally, a computation is *deterministic* provided that as the computation unfolds for any input, at each step in the computation, there is at most one "next step." On the other hand, for *non-deterministic* computations, there may be multiple "next steps" in the course of the computation. Multiple output forms would seem to imply that at some point in the computation there need be multiple "next steps."

In the case of the automata we have looked at so far, there was at most one initial state. Also, the delta function had at most one output. Consequently, there was at most one path through any given automata for a particular input string or tree.

In this section, we study what happens if we add non-determinism into the picture. In other words, we will effectively change the delta *function* to a *relation* and allow more than one initial state.

As it turns out, in terms of the class of recognizable stringsets and treesets, the addition of non-determinism has little consequence. However, for the classes of string and tree transductions, the effects of non-determinism are significant.

### 6.1.1 Non-deterministic string acceptors

**Definition 31.** *A* non-deterministic finite-state acceptor (NFA) *is a tuple* $(Q, \Sigma, I, F, \delta)$ *where*

- $Q$ *is a finite set of states;*
- $\Sigma$ *is a finite set of symbols (*the alphabet*);*
- $I \subseteq Q$ *is a set of* initial *states;*
- $F \subseteq Q$ *is a set of* accepting *(*final*) states; and*
- $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times Q$*. It is the* transition relation.

The symbol $\epsilon$ denotes a "free" change of state; that is the state of the system can change without any input being consumed.

The *epsilon closure* of a set of states $S \subseteq Q$ is the smallest subset $S'$ of $Q$ satisfying these properties:

1. $S \subseteq S'$
2. For all $q, r \in Q$: if $q \in S'$ and $r \in \delta(q, \epsilon)$ then $r \in S'$.

The epsilon closure of $Q$ is denoted $C_\epsilon(Q)$.

$\delta$ is a relation and in order to define the "process" function, we use $\delta$ to define related functions. First, we define $\delta' : Q \times \Sigma \rightarrow \wp(Q)$. Recall that for any set $A$, $\wp(A)$ denotes the powerset of $A$, which is the set of all subsets of $A$.

$$\delta'(q, a) = \{q' \mid (q, a, q') \in \delta\}$$

Next we define $\delta'' : \wp(Q) \times \Sigma \rightarrow \wp(Q)$ as follows.

$$\delta''(Q, a) = \bigcup_{q \in Q} \delta'(q, a)$$

Finally, we can explain how a NFA processes strings using the recursive definition below.

$$\begin{aligned} \pi(Q, \lambda) &= Q \\ \pi(Q, aw) &= \pi(C_\epsilon(\delta''(Q, a)), w) \end{aligned} \tag{6.1}$$

Consider some DFA $A = (Q, \Sigma, I, F, \delta)$ and string $w \in \Sigma^*$. If $\pi(C_\epsilon(I), w) \cap F \neq \emptyset$ then we say $A$ *accepts/recognizes/generates* $w$. Otherwise $A$ *rejects* $w$.

**Definition 32.** *The stringset recognized by $A$ is $L(A) = \{w \in \Sigma^* \mid \pi(C_\epsilon(I), w) \cap F \neq \emptyset\}$.*

**Exercise 19.**

1. Non-determinism makes expressing some stringsets easier because it can be done with fewer states.

   (a) Write an acceptor for the set of strings where the last letter must be a consonant.

   (b) Write an acceptor for the set of strings where the second-to-last letter must be a consonant.

   (c) Write an acceptor for the set of strings where the third-to-last letter must be a consonant.

2. Use the plurality of initial states to show that if $R$ and $S$ are stringsets each recognized by some NFA that the union $R \cup S$ is also recognized by a NFA.

3. Use epsilon transitions to show that if $R$ and $S$ are stringsets each recognized by some NFA that the concatenation $RS$ is also recognized by a NFA.

**Theorem 15.** *The class of stringsets recognizable by the NFA are exactly the regular stringsets.*

This theorem states that there is no gain in expressivity by introducing non-determinism into the models. However, as we have seen there can be gains practically in terms of how we write the models.

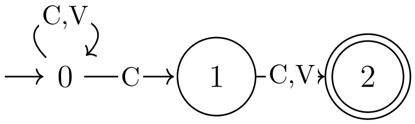
The proof of the above theorem shows how, for any NFA with state set $Q$, to construct an equivalent DFA. The construction is often called *the subset construction* because the states in the DFA will correspond to subsets of $Q$.

Here is the basic idea (ignoring epsilon transitions.) Consider any NFA $N = (Q_N, \Sigma, I_N, F_N, \delta_N)$. Then we can construct an equivalent DFA $D = (Q_D, \Sigma, I_D, F_D, \delta_D)$ as follows.

1. $Q_D = \wp(Q_N)$
2. $I_D = I_N$
3. $F_D = \{S \mid S \cap F_N \neq \emptyset\}$
4. $\delta_D(S, a) = \bigcup_{q \in S} \delta_N(q, a)$

Essentially, sets of states in the NFA correspond to distinct states in the equivalent DFA. (If the NFA includes epsilon transitions, it will be important to take into account the epsilon closure of $S$ and the states reachable from $S$ on $a$ in the calculation $\delta_D(S, a)$.)

Here is an example. Consider the NFA below.

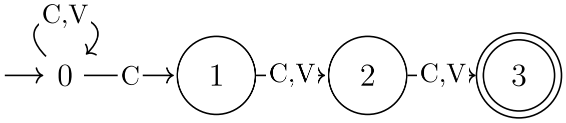


Here is the full transition table according to the subset construction. Final states are marked with the * and the initial state with →.

| $Q_D$ | C | V |
|---|---|---|
| ∅ | ∅ | ∅ |
| →{0} | {0,1} | {0} |
| {1} | {2} | ∅ |
| *{2} | ∅ | ∅ |
| {0,1} | {0,1,2} | {0,2} |
| *{0,2} | {0,1} | {0} |
| *{1,2} | {2} | {2} |
| *{0,1,2} | {0,1,2} | {0,2} |

**Exercise 20.** Let's draw what this looks like. Which states are useless? If the useless states are removed, is the resulting machine complete?

**Exercise 21.** Write the DFA equivalent to the NFA shown below.



### 6.1.2 Tree acceptors

**Theorem 16.** *The class of treesets recognized by non-determinstic bottom-up acceptors is exactly the recognizable treesets (so the class of treesets recognized by determinstic bottom-up acceptors).*

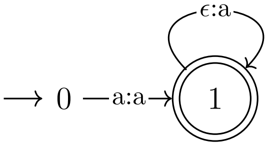
**Theorem 17.** *The class of treesets recognized by non-determinstic top-down acceptors is also exactly the recognizable treesets.*

### 6.1.3 Non-deterministic string transducers

**Theorem 18.** *The left projection (domain) of a regular relation is a regular stringset. The right projection (co-domain) of a regular relation is a regular stringset.*

**Theorem 19.** *Left and right sequential functions are proper subclasses of the relations recognized by non-deterministic string transducers.*

One reason why this is true is that epsilon transitions permit non-deterministic transducers to recognize *infinite relations* not functions. For example, the non-deterministic acceptor below relates input $a$ to every element in $\{a, aa, aaa, \ldots\}$.



This is not possible for sequential transducers with the $\Sigma^*$ monoid or the finite language monoid. However, it should be possible with the "Regular language" monoid where the outputs of the transitions are actually NFA that get "concatenated."

### 6.1.4 Tree transducers

Recall that a tree transducer is *linear* when the variable trees in the omega function include at most one of each variable. (In other words, trees are not copied.)

**Theorem 20.** *The domain of a non-deterministic tree transducer is a recognizable tree language. The image of a recognizable tree language by a linear tree transducer is recognizable.*

**Theorem 21.** *Generally, the class of tree relations recognized by non-deterministic top-down tree transducers and non-deterministic bottom-up tree transducers are incomparable.*

**Theorem 22.** *However, any linear top-down tree transducer is equivalent to a linear bottom-up tree transducer.*

### 6.1.5 Summary

For acceptors recognizing sets of strings or trees, non-determinism does NOT entail an increase in expressivity as the subset construction shows. However, non-determinstic computations can require exponentially fewer states.

For transducers, the current understanding is that non-determinism DOES lead to more expressive constructions. The ability of transducers to recognize inifinite relations is usually taken to be the key example. However, there are examples that do not require infinite relations as discussed in Heinz and Lai (2013).

There is much more to cover here more generally, see Hopcroft *et al.* (2001) and Comon *et al.* (2007)

# Bibliography

Applegate, R.B. 1972. Ineseño Chumash grammar. Doctoral dissertation, University of California, Berkeley.

Beesley, Kenneth, and Lauri Kartunnen. 2003. *Finite State Morphology*. CSLI Publications.

Choffrut, Christian. 2003. Minimizing subsequential transducers: a survey. *Theoretical Computer Science* 292:131 – 143.

Chomsky, Noam. 1956. Three models for the description of language. *IRE Transactions on Information Theory* 113–124. IT-2.

Comon, H., M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. 2007. Tree automata techniques and applications. Available on: `http://tata.gforge.inria.fr/`. Release October, 12th 2007.

Davis, Martin D., and Elaine J. Weyuker. 1983. *Computability, Complexity and Languages*. Academic Press.

Graf, Thomas. 2011. Closure properties of Minimalist derivation tree languages. In *LACL 2011*, edited by Sylvain Pogodalla and Jean-Philippe Prost, vol. 6736 of *Lecture Notes in Artificial Intelligence*, 96–111. Heidelberg: Springer.

Graf, Thomas. 2015. Computational unity across language modules. Invited talk, December 15, Department of Theoretical and Applied Linguistics, Moscow State University, Moscow, Russia.

Harrison, Michael A. 1978. *Introduction to Formal Language Theory*. Addison-Wesley Publishing Company.

Heinz, Jeffrey. 2018. The computational nature of phonological generalizations. In *Phonological Typology*, edited by Larry Hyman and Frans Plank, Phonetics and Phonology. Mouton. Final version submitted November 2015. Expected publication in 2017.

Heinz, Jeffrey, and Regine Lai. 2013. Vowel harmony and subsequentiality. In *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, edited by Andras Kornai and Marco Kuhlmann, 52–63. Sofia, Bulgaria.

de la Higuera, Colin. 2010. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press.

Hopcroft, John, Rajeev Motwani, and Jeffrey Ullman. 2001. *Introduction to Automata Theory, Languages, and Computation*. Boston, MA: Addison-Wesley.

Jardine, Adam. 2016. Computationally, tone is different. *Phonology* 32:247–283.

Johnson, C. Douglas. 1972. *Formal Aspects of Phonological Description*. The Hague: Mouton.

Jurafsky, Daniel, and James Martin. 2008. *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics*. 2nd ed. Upper Saddle River, NJ: Prentice-Hall.

Kaplan, Ronald, and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics* 20:331–378.

Kobele, Gregory M. 2011. Minimalist tree languages are closed under intersection with recognizable tree languages. In *LACL 2011*, edited by Sylvain Pogodalla and Jean-Philippe Prost, vol. 6736 of *Lecture Notes in Artificial Intelligence*, 129–144. Berlin: Springer.

Lothaire, M., ed. 2005. *Applied Combinatorics on Words*. 2nd ed. Cmbridge University Press.

Mohri, Mehryar. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics* 23:269–311.

Mohri, Mehryar. 2005. Statistical natural language processing. In *Applied Combinatorics on Words*, edited by M. Lothaire. Cambridge University Press.

Roark, Brian, and Richard Sproat. 2007. *Computational Approaches to Morphology and Syntax*. Oxford: Oxford University Press.

Roche, Emmanuel, and Yves Schabes. 1997. *Finite-State Language Processing*. MIT Press.

Rogers, James. 1998. *A Descriptive Approach to Language-Theoretic Complexity*. Stanford, CA: CSLI Publications.

Shieber, Stuart. 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy* 8:333–343.

Sipser, Michael. 1997. *Introduction to the Theory of Computation*. PWS Publishing Company.